UNIVERSITY OF PLYMOUTH
School of Engineering, Computing & Mathematics

Session 3: Objects

Full-Stack Development

Mark Dixon

School of Engineering, Computing and Mathematics

Last Week

- What did we do last week (write down topics, what you can remember)?
  - Did you learn anything (was anything particularly useful or good)?
  - How did the lab session go (were you able to get something running)?
  - What could be better?

Introduction

Today's topics

1. Characteristics of objects and their use
2. Defining your own objects
3. JSON
4. Ajax

Session learning outcomes – **by the end of today's lecture you will be able to:**

- Use objects to represent data within your own JavaScript programs
- Write your own bespoke objects to represent data arising within your programs

Object-oriented programming

The solution to a problem is

- A collection of objects with individual attributes and responsibilities
- Objects interact through “message passing”

Advantages

- Encapsulation/information hiding
- Intuitive
- Reuse

Weaknesses

- Can be overkill for limited applications
- Can lead to spaghetti code

Objects

An unordered collection of data with associated keys

- Objects have **methods**, **inherit** from other objects and are **dynamic**

Objects attributes

- Object prototype (references the object properties are inherited from)
- class (a string categorising the object type)
- Extensible flag (specifies whether new properties can be added)

Object categories

- Native object (defined by the browser – arrays, functions, dates…)
- Host object (e.g., HTML elements)
- User-defined objects (created by executing JS code)

Properties & values

Properties

- Unique identifier for values
- Any string (including the empty string)
- Property attributes
  - Writeable
  - Enumerable
  - Configurable

Values

- Any valid JavaScript value, or a method

JavaScript objects

Objects have keys and values (associative arrays, hashmaps)

```
person = {
    name: "Barry",
    age: 42,
}
```

Dot notation for access

```
person.email = "barry@plymouth.ac.uk"
```

You can also loop through object properties

```
for (property in person) {
    if (person.hasOwnProperty(property)) {
        console.log(property + ": " + person[property]);
    }
}
```

JavaScript strings

```
let str = "Hello World";
```

- JavaScript strings are a collection of characters
- No char type – just a string with a single character
- All strings have a property called length
- Strings can be created as objects using

```
new String("Hello World");
```

but this is much slower than using primitive strings

Regular expressions

/pattern/modifier

- **Pattern** – the pattern to be matched
- **Modifier** –
  - **i** – case-insensitive matching
  - **g** – find all matches
  - **m** – multi-line matching
- **Character classes and groups** – [abc] and (ab|cd)
- **Quantifiers** – specify how many times a character or pattern should appear (e.g., +, *, …)
- **Predefined character classes** (e.g., \d, \s, \D…)

JavaScript arrays

Declaration

```
things = [“Barry”, 42, “barry@plymouth.ac.uk”]
```

Loop

```
for (index=0; index<things.length; index++) {
    text += things[index];
}
```

Extra properties

- length, sort(), push(), indexOf(“Barry”);

Object orientation concepts

Aggregation

- A link to another object

Classes and instances

- Variable instantiation
- Shared variables

Inheritance

- Code reuse
- Multiple inheritance
- Polymorphism

Information hiding

JavaScript approaches to object orientation

Implement OO with JS features yourself

- Use prototypes and instances as classes and objects

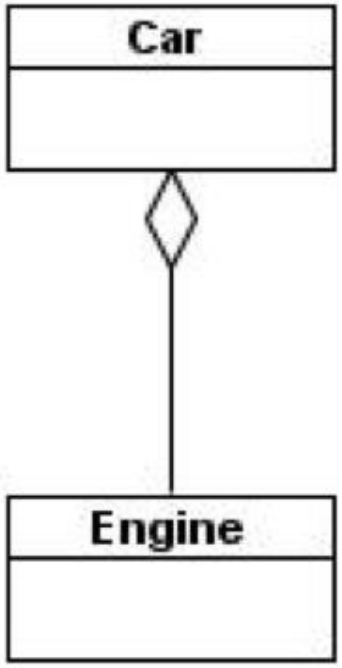
Using a third party library/language

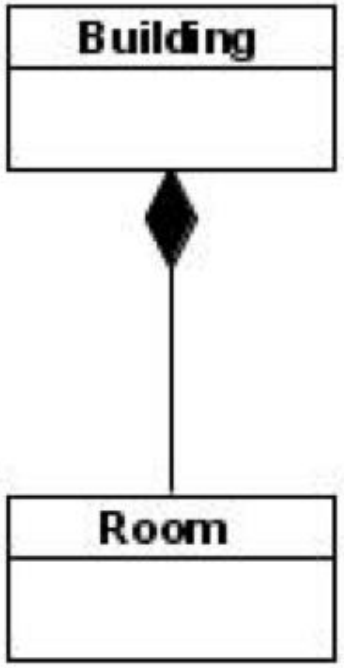
- **TypeScript** – supports full OO features and syntax
  - types, class, extends, interface

Rethink OO

- Question the value of individual OO concepts

Aggregation & Composition






```
let myEngine = {
    type: "diesel",
    volume: 2.1,
    weight: 35
}

let car = {
    type: "fiat",
    engine: myEngine,
    colour: "green"
}

console.log(car.engine.type);
```

Engine can exist without Car (Aggregation)

Variable instantiation – named functions

- JavaScript functions can be used as **constructors**
- Function instantiations work as objects

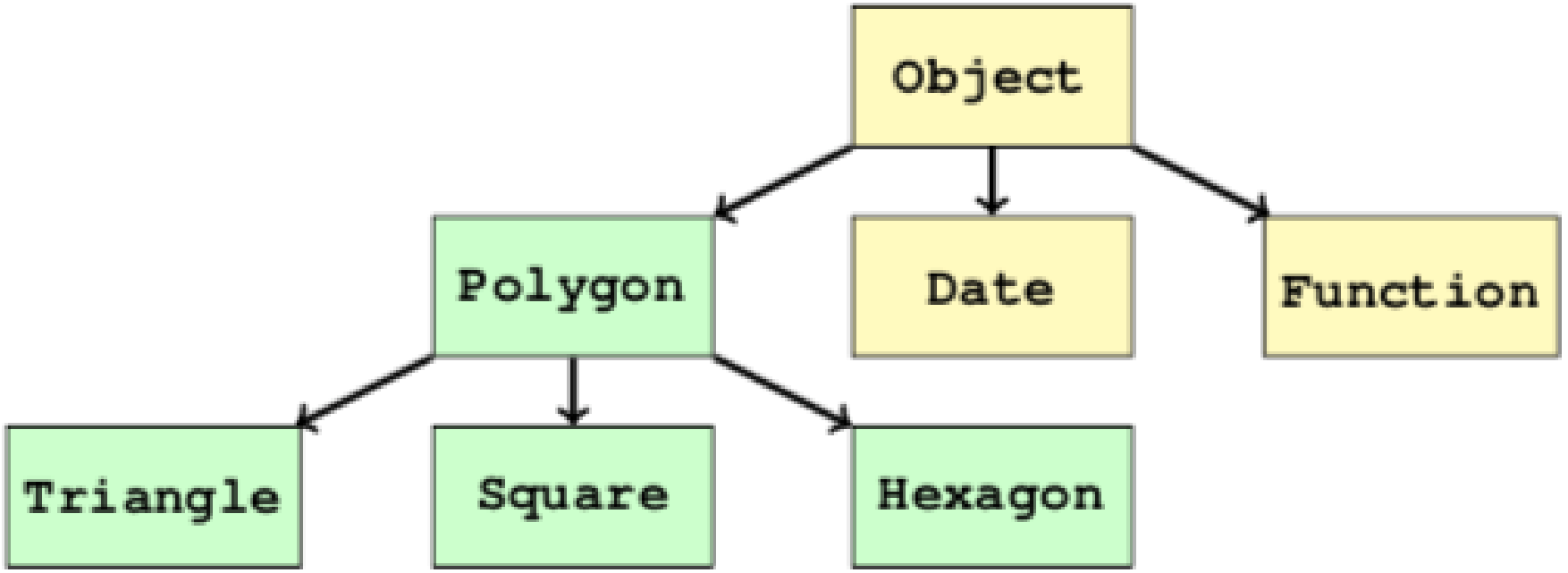
```
function World(population) {
    this.population = population;
    this.rotation = 1;

    this.rotate = function(d) {
        this.rotation = this.rotation + d;
    }
}

let earth = new World(5000000000)
earth.rotate(90)

let mars = new World(0);
```

Function prototypes

When a JS object is declared it creates a related prototype data structure containing important function parameters and sub-functions

- Name (may be empty), number of arguments, call, apply
- Accessible through the instance variable “prototype”

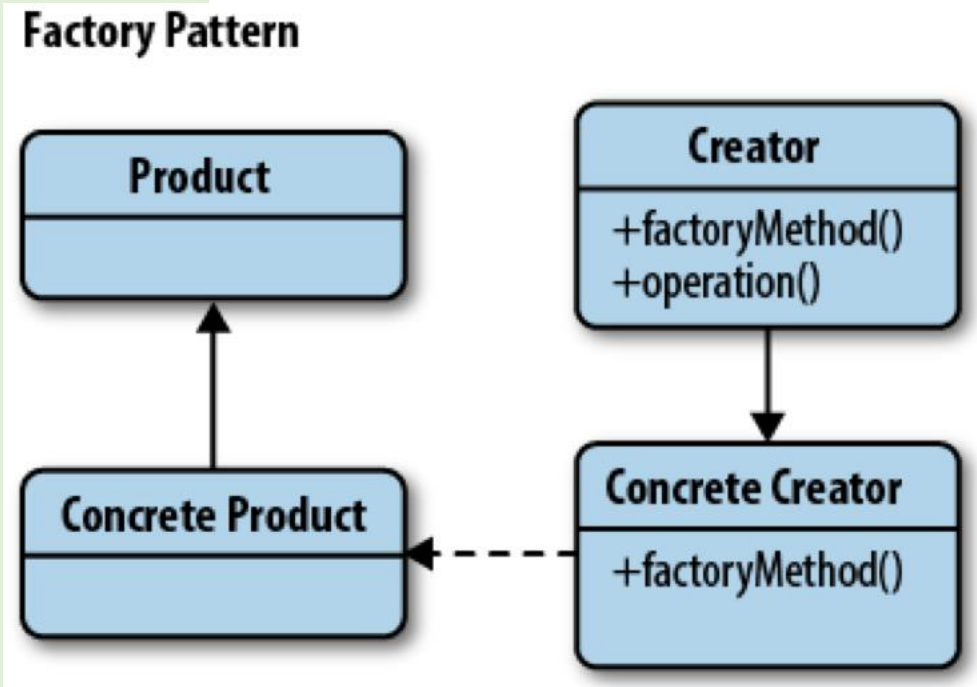
Prototypes are shared between all instances



Variable instantiation - Factory

Factory design pattern – creates an object for producing other objects

```
factory = {
    create: function(product, ptype) {
        let p;
        if (product == "car") {
            type: ptype,
            engine: "diesel",
            colour: "green"
        } else {
            p = {}; // Other objects
        }
        return p;
    }
}

let myFiat = factory.create("car", "fiat");
let myVolvo = factory.create("car", "volvo");
```



Inheritance

Allows a class to access methods from another class – do not repeat yourself!

```
function Rectangle(width, height) {
    this.width = width;
    this.height = height;
}

Rectangle.prototype.computeArea = function() {
    return this.width * this.height;
}

function Square(width) {
    Rectangle.call(this, width, width);
}

Square.prototype = Object.create(Rectangle.prototype);
```

Criticism of classical inheritance

“Favour object composition over class inheritance”

Gamma, Helm, Johnson, Vlissides

“The problem with object-oriented languages is they’ve got all this implicit environment that they carry around with them. You wanted a banana, but what you got was a gorilla holding the banana and the entire jungle”

Joe Armstrong

Overriding

Provides **new code** for an **existing superclass method**

```
function Rectangle(width, height) {
    this.width = width;
    this.height = height;
}

Rectangle.prototype.computeArea = function() {
    return this.width * this.height;
}

function Square(width) {
    Rectangle.call(this, width, width);
}

Square.prototype = Object.create(Rectangle.prototype);

Square.prototype.computeArea = function() {
    return Math.pow(this.width, 2);
}
```

Extending a superclass

Uses a superclass method and adds functionality

```
Rectangle.prototype.computeArea = function() {
    return this.width * this.height;
}

Square.prototype.computeArea = function() {
    console.log("Do something new here");
    return Rectangle.prototype.computeArea.call(this);
}
```

Polymorphism

- A single interface to entities of different types
- Superclass can be used for array membership
- Trivial in JavaScript with no type checking

```
$(function() {
    arr = [];
    arr[0] = new Rectangle(5, 12);
    arr[1] = new Square(6);

    for (i=0; i<arr.length; i++) {
        console.log(arr[i].computeArea());
    }
);
```

Information hiding

- The closure of the construction function is only accessible to the functions defined within the constructor function
- Variables local to the constructor function can be used as private variables

```
function Counter(start) {
    let count = start;

    return {
        increment: function() {
            count++;
        },
        get: function() {
            return count;
        }
    }
}

let foo = Counter(4);
```

Extending the class

```
class Square extends Rectangle {
    constructor(width) {
        super(width, width);
    }
}

$(function() {
    let squareInstance = new Rectangle(4);
    console.log(squareInstance.computeArea());
});
```

- Specify the class being extended with the **extends** keyword
- Reference the parent class with **super**

JSON

JavaScript **Object** Notation

```
{ "modules":
    [
        {"code": "COMP2002",
         "name": "Artificial Intelligence",
         "stage": 2},

        {"code": "COMP3006",
         "name": "Full-Stack Development",
         "stage": 4},
    ]
}
```

- Serialisable data that can be converted directly into a JS object in your program
- Widely used for data exchange online

JSON

JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages (including C, C++ and C#), Java, JavaScript, Perl, Python, and many others

It is based on JavaScript object declarations

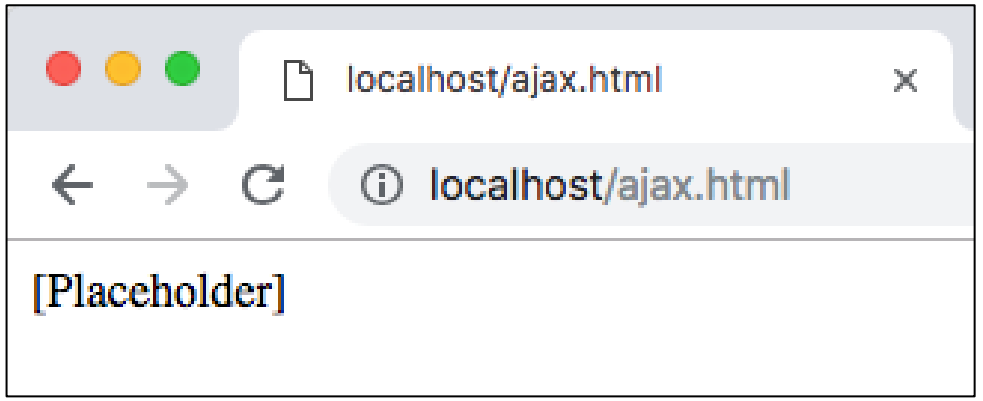
- Keys
- Values

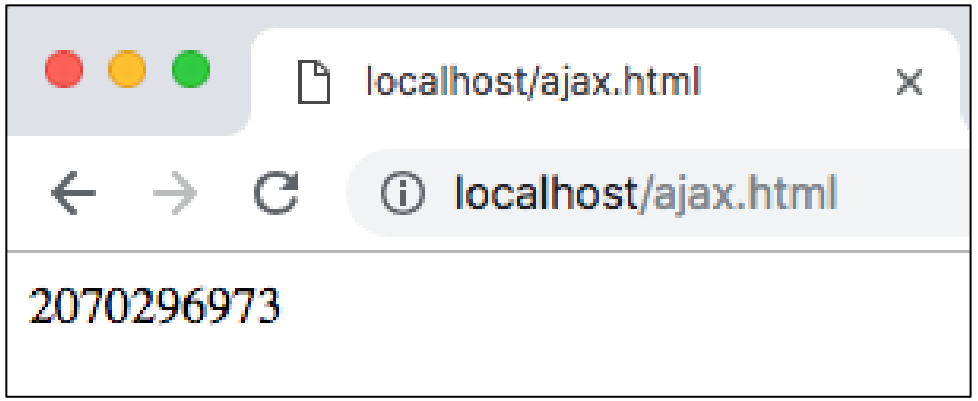
```
{ "name": "Barry", "age": 42, "email": "barry@plymouth.ac.uk" }
```

Convert the object to a string so that it can be stored in a file or transmitted across the internet

Ajax

Ask a server for a random number using an AJAX call and insert the resulting value into a webpage






```
<html>
    <head>
        <script src="ajax.js"></script>
    </head>
    <body>
        <p id="randnum">[Placeholder]</p>
    </body>
</html>
```

Ajax – vanilla / raw JS

1. Initialise a new **XMLHttpRequest** object
2. Set up an event handler for the object
3. Send the requ

```
window.onload = function() {
    // Initialise a new XMLHttpRequest object.
    let ajaxObj = new XMLHttpRequest();

    // Define the event handler
    ajaxObj.onreadystatechange = function() {
        if (this.readyState === 4 && this.status == 200) {
            let elem = document.getElementById("randnum");
            elem.innerHTML = this.responseText;
        }
    };

    // Send the request.
    ajaxObj.open("GET", "http://localhost/rand.php", true);
    ajaxObj.send();
};
```

The jQuery equivalent…

Use the get function

- **$.get(…)** – arguments are the **URL** and an **event handler**

```
$(function() {
    $.get("http://localhost/rand.php", function(data) {
        $("#randnum).html(data);
    });
});
```

How does the get function work

- **$.ajax()** – jQuery's base Ajax function, very powerful
- **get** function calls base function
- **wrapper** functions (such as get) usually sufficient

Error handling with Ajax

```
if (ajaxObj.readyState === 4) {
    switch(ajaxObj.status) {
        case 200: // All is well, do the standard thing.
            document.getElementById("randnum").innerHTML = text;
            break;

        case 404: // Page not found - report it.
            let error = "404: " + url + " " + ajaxObj.statusText;
            document.getElementById("error").innerHTML = error;
            break;

        case 500: // Server error - report it.
            let error = "500 " + ajaxObj.statusText;
            document.getElementById("error").innerHTML = error;
            break;
    }
}
```

Ajax – advantages and disadvantages

Advantages

- Allows webpage content to be updated without page reload
- Based on web standards (such as JavaScript, XML…)
- Asynchronous communication between client & server

Disadvantages

- Security – data is sent using HTTPS
- The user cannot tell that content is loading
- The page may not “exist” for browser navigation and web crawlers

Summary

Objects

- Maps keys to values
- Provided by the language, browser, and can be hand-rolled
- The basis for JSON

Defining objects

- Multiple different ways of defining objects
- Can be done using classes since ECMAScript 2015